Arithmetic Issues in Geometric Computations

D. Michelucci
Ecole des Mines, F-42023 Saint-Étienne 02
micheluc@emse.fr

Abstract

This paper first recalls by some examples the damages that the numerical inaccuracy of the floating-point arithmetic can cause during geometric computations, and it intends to explain why damages for geometric computations differ from those met in numerical computations. Then it surveys the various approaches proposed to overcome inaccuracy difficulties; conservative approaches use classical geometric methods but with 'exotic' arithmetics instead of the standard floating-point one; radical ones go farther and reject classical techniques, considering them not robust enough against inaccuracy.

1 Introduction

Geometric modellers provided by commercial CADCAM softwares, and methods from the more theoretical field of *Computational Geometry* all perform geometric computations: for instance triangulating or meshing geometric domains for finite elements simulation, or computing intersections between geometric objects. Inaccuracy is a crucial issue for geometric computations. Not only the numerical results can be inaccurate, but also geometric programs can crash, or enter in infinite loops, or terminate but provide inconsistent results, the topology of which is the topology of no possible geometric objects.

Today it is widely known that numerical analysis can suffer from numerical inaccuracy, especially in presence of ill-conditionning. It is perhaps less known that the situation is even worse for geometric algorithms. Due to the underlying combinatoric properties of geometric objects, numerical inaccuracy can cause damages even when there is no ill-conditionning at all, as this paper will show by some typical examples.

To solve or bypass inaccuracy problems in geometric computations, several approaches have been explored. Conservative approaches use classical techniques but with 'exotic' arithmetics, *ie* they do not rely only on the standard floating-point arithmetic. More radical approaches have recently proposed to get rid of some classical methods or data structures because they do not withstand inaccuracy, namely methods from computational geometry and topology-based data structures like BReps (*Boundary representations*). Thus taking inaccuracy into account can radically modify methods and data structures used in geometric computing.

This paper only deals with the inaccuracy problem, but the latter is not the only arithmetic issue for geometric computations: there are two others. The first problem is due to the overwhelming number of degenerate geometric cases (say: alignment of more than two points, coplanarity of more than three points, cocircularity of more than three points, intersection of more than two lines in a point, parallelism between lines, etc) which geometric methods have to handle and the programmer has to treat: it is not obvious that this is an arithmetic problem, but a solution is an arithmetic one: it symbolically perturbs the data by infinitely small values to remove degeneracies [EM90, EC92, Mic95], using a non-standard arithmetic (*ie* an arithmetic computing with non-standard, infinitely small numbers). However, this arithmetic solution has the drawback of needing an exact arithmetic. The second arithmetic issue for geometric computations is that data acquired from some sensors, or mechanical products machined by imperfect tools, are also known only up to some finite precision: some CADCAM applications need to take into account this other kind of inaccuracy [Jus92]. These two issues are beyond the scope of this paper.

Plane of this paper. Section 2 explains the typical damages due to inaccuracy: after section 2.1 fixes notations, section 2.2 shows how a method from Computer Graphics infinitely loops under inaccuracy, section 2.3 details running-time crashs of a method from Computational Geometry under numerical imprecision, section 2.4 a topological algorithm which may also infinitely loop. These failures are always due to the fact that inaccuracy contradicts geometric properties. Section 2.5 presents other examples of geometric properties invalidated by inaccuracy and section 2.6 concludes this part about imprecision ravages on geometric algorithms. Section 3 surveys the different approachs investigated to fight against (or to avoid) inaccuracy: section 3.1 presents the popular ϵ heurism, section 3.2 some empirical rules of programming that avoid the more obvious geometric inconsistencies, section 3.3 the spirit of the approach which tries to always provide consistant –though approximated– results, section 3.4 some typical exact arithmetics, section 3.5 an interval confining approach which extents the ϵ heurism, and section 3.6 a new tendency emerging in the CADCAM and Computer Graphic fields which rejects methods and data structures considered not robust enough against imprecision. Section 4 concludes.

2 Some damages of numerical inaccuracy

2.1 Notations

We will use the following notations :

1. Straight lines with equation $\alpha x + \beta y + \gamma = 0$ are represented by a triple of floating-point numbers (α, β, γ).
2. The triple for the line through two distinct points A and B is

$$(y_B - y_A, x_A - x_B, x_B y_A - x_A y_B);$$

3. The intersection between two given lines $D : (\alpha, \beta, \gamma)$ and $D' : (\alpha', \beta', \gamma')$ is the point $(x_\Omega,\ y_\Omega)$ with
$$x_\Omega = \frac{\beta\gamma' - \beta'\gamma}{\alpha\beta' - \alpha'\beta}, \quad y_\Omega = \frac{\alpha'\gamma - \alpha\gamma'}{\alpha\beta' - \alpha'\beta}$$
4. Geometric algorithms often use the *lexicographical order* on coordinates, defined by:
$$(x, y) <_L (x', y') \Leftrightarrow x < x' \text{ or } (x = x' \text{ and } y < y').$$

A line with $\beta = 0$ (respectively $\alpha = 0$) *ie* parallel to the Oy (respectively Ox) axis is said to be vertical (respectively horizontal).

The notation a^* denotes the floating-point approximation of a. *fp* is a shortcut for 'floating-point'. $\lfloor x \rfloor$ stands for x floor, and $\lceil x \rceil$ for x ceil.

2.2 Consequences on an algorithm from Computer Graphics

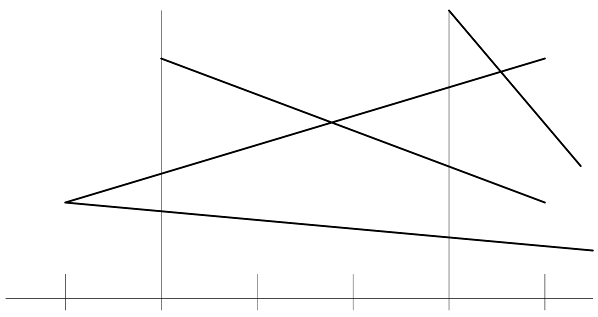

Figure 1: *A span, delimited by two vertical lines.*

This section studies the consequences of inaccuracy on a $2D$ algorithm. The data are segments. For all lines L_x parallel to the Oy axis and having integer abscissa: $x \in \{0, 1, 2 \dots N\}$, the problem is to find the segments crossing L_x and to sort them by increasing ordinate. This problem occurs in Computer Graphics as a sub-problem in Atherton's method, which computes images of $3D$ scenes defined by boolean combinations (intersection, union and difference) of polyedra.

The standard method first determines the set of *spans*: a span is a maximal interval $[x_0, x_1]$ such that $]x_0, x_1[$ contains no initial vertex, but possibly contains intersection points. This stage may be achieved for instance by sorting all endpoints by increasing abscissa, then by scanning the resulting sorted list V of vertices while maintaining the set S of active segments: if $v_j = (x_j, y_j)$ is the left (respectively the right) vertex of the segment s_j, insert s_j in S (respectively remove s_j from S) so that S is still the set of the active segments in the interval $]x_j, x_{j+1}[$. It is also possible to use some bucket-sort scheme instead.

Let S be the set of the active segments in the current span $]x_0, x_1[$. Let S_0 (respectively S_1) be the result of sorting S by increasing ordinate in $x = x_0$ (respectively in $x = x_1$). If S_0 equals S_1 we are done and we study the next span, otherwise let i be the lowest index such that $S_0[i] \neq S_1[i]$. Then these two segments $S_0[i]$ and $S_1[i]$ cut each other somewhere in $[x_0, x_1]$, say in (x_i, y_i). In such a case, recursively study the span $]x_0, \lfloor x_i \rfloor[$ and $]\lceil x_i \rceil, x_1[$. At the end, either there is no intersection points inside the span, or $x_0 = x_1$.

Though correct, this method infinitely loops because of inaccuracy, or at least it loops until the stack is filled up by recursive calls: It happens that the computed intersection point has *fp* abscissa x_i outside the interval $[x_0, x_1]$, say between x_1 and $x_1 + 1$ (of course, this situation is absurd and cannot occur without inaccuracy). Thus the procedure `span`(x_0, x_1) recursively calls `span`$(x_0, x_1)\ldots$: an infinite loop.

This problem is trivially solved: when x_i is greater than x_1, cut $[x_0, x_1]$ into $[x_0, x_1 - 1]$. Symmetrically when x_i is lower than x_0. It is worth noting that the reliability against inaccuracy of programs from Computer Graphics is much more easily achieved than for the ones from Computational Geometry.

2.3 Consequences on a Bentley and Ottman's algorithm

This section details the consequences of inaccuracy on a classical and representative algorithm from Computational Geometry, the Bentley and Ottman's method.

2.3.1 Bentley and Ottman's algorithm

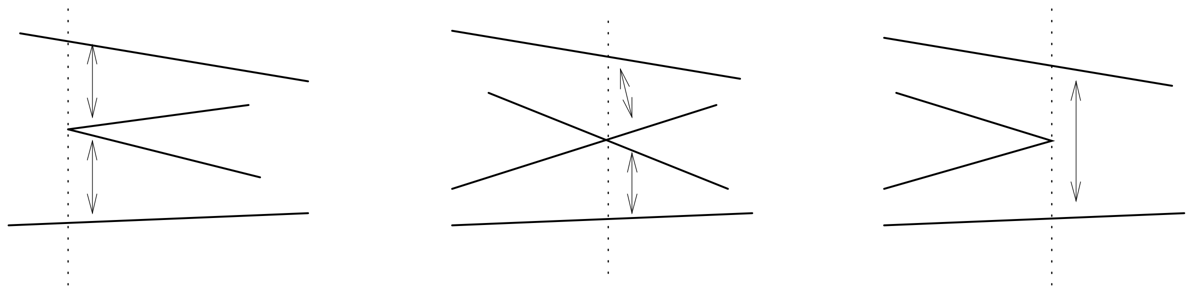

Figure 2: *The situations where two segments become contiguous along the sweeping line. The vertical dotted line represents the sweeping line, the arrows show the couple of contiguous segments.*

In a nutshell, this algorithm computes all the k intersection points between n segments in the plane, in time $O((n + k) \log n)$. For sake of simplicity, this exposition will ignore degeneracies: intersection points common to more than two segments, vertical segments, intersection points confused with initial vertices, and so on. The principle is to sweep the plane by a vertical line from left to right, *ie* initial vertices are swept in lexicographic order, and to maintain the set of the segments crossing the sweeping line, ordered upwards. The method exploits the two following remarks: first, the order of the segments crossing the sweeping line obviously depends on the abscissa of the latter, but it changes only locally when passing an initial vertex or an intersection point: local changes are insertion or deletion of a segment when passing a vertex, or permutation of two intersecting segments when passing an intersection point. Secondly, two segments can cross each others only after they have been contiguous along the sweeping line (ignoring degeneracies): thus, if each time two segments become contiguous along the sweeping line the algorithm checks their possible intersection, then all intersection points will be found. The three situations where two segments become contiguous along the vertical sweeping line are shown in Fig. 2: when a vertex is the beginning (*ie* the left vertex) of one or several segments; when a vertex is the end (*ie* the right vertex) of one or several segments, or when a vertex is simultaneously an end and a beginning, for instance an intersection point.

The algorithm is as follows. Set X to all initial vertices: X is ordered by the lexicographic order. Let Y be the set of segments crossing the current sweeping line. Y is ordered upwards. X and Y may be represented by balanced trees. Initially Y is empty. While X is not empty do: Let p be the first point of X, and remove p from X. Let b and a be the segment just below and just above e. If there are some segments with p as right endpoint, remove them from Y. If p is the left vertex of some segments $s_1 \dots s_t$ (ordered by increasing slope), insert them in Y, then compare b with s_1 and a with s_t, since b and s_1 on one hand and a and s_t on the other hand become contiguous in Y. Otherwise, when there are no segments with left vertex p, a and b become contiguous in Y and thus are compared. Each time a new intersection point is found, insert it in X and cut the corresponding segments.

2.3.2 A special configuration

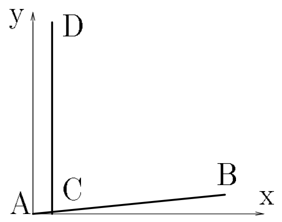

Figure 3: *A special configuration of two segments.*

M. Gangnet communicated me the configuration of segments in Fig. 3 in 1983. Assume for convenience that a *fp*-arithmetic with base 10 is used, with 4 significative digits. A is the origin, B has coordinates (10., 1.), C has coordinates (1., 0.) and last, D has coordinates (1. + u, 10.). We will see that for $u \in]-0.1, +0.1[$, *ie* for all points D_u

in the open segment $](0.9, 10.), (1.1, 10.)[$, all *fp* intersection points $I_u^* = (AB \cap CD_u)^*$ have coordinates $I_0 = I_0^* = (1., 0.1)$: the coefficients triple for line AB is (1, 10, 0), the one of CD_u is $(10, -u, -10)$, thus $\Delta x = \Delta x^* = -100$, $\Delta y = \Delta y^* = 10$, but $\Delta = -(100 + u)$ is represented in the best case by the *fp* value $\Delta^* = 100.$ for all u in the open interval $]-0.1, +0.1[$ (remember we only have 4 digits: due to cancelation, $100.0 + (u = 0.07) = 100.0$!).

As a first consequence, if a configuration contains segments say $CD_{-0.099}$, CD_0, $CD_{0.099}$ on one hand and AB on the other hand, the *fp* arithmetic cannot represent it in a consistant way – and no matter the algorithm used to compute the intersection points: points $I_{-0.099}$, I_0 and $I_{0.099}$ have the same *fp* coordinates, whereas $CI_{-0.099} \subset CD_{-0.099}$, $CI_0 \subset CD_0$, $CI_{0.099} \subset CD_{0.099}$ are different segments.

Of course, this *fp* arithmetic may seem unrealistic, it has been chosen only to simplify the statement. It is possible to find similar configurations for all *fp* arithmetic, no matter the used base and the length of the mantissa. We now follow the behaviour of the Bentley-Ottman's method on such a configuration.

2.3.3 Consequences

Consider the configuration of two segments AB and D_uC, with $u < 0$ and u in the inconsistant interval, say $u = -0.05$. From now on, we just use D for D_u and I for I_u. To compute if the point $I^* = (AB \cap CD)^* = (1., 0.1)$ belongs to the segment DC, two tests are possible.

First, $I^* \in CD$ iff $D \leq_L I \leq_L C$: with this test (mathematically correct when there is no inaccuracy), I^* does not belong to CD. A first mistake: the intersection point I^* is forgotten.

A second test, mathematically equivalent to the first one when there is no inaccuracy, is: $I^* \in CD$ iff $x_D \leq x_I \leq x_C$ and $y_C \leq y_I \leq y_D$, taking into account that DC has a negative slope. Here this test will correctly conclude that I^* belongs to CD, in contradiction with the previous and theoretically equivalent test. Note it is possible to find other examples where this last test fails.

• **If I^* is forgotten**. Suppose first that the first test is used, or any test such that I^* is forgotten. Thus when the sweeping line passes C, the program believes that (or in less anthropomorphic words, the data structure Y stores that) AB is below CD, which is wrong. Since C is the right endpoint of DC, the program has to remove DC from Y. When Y is the *only* data structure accessing segments, as it is the case in the original Bentley and Ottman's paper, a classic binary search through Y is needed to find DC in Y, starting from the Y root. We can suppose that the left (respectively the right) subtree stores the segments below (respectively above) the one of the root. Here the program wrongly believes that AB is below DC in C, thus for instance the root carries segment AB, and its right son carries segment DC. To find DC, the program compares the height of the searched DC segment and the one of the root segment AB, *ie* it computes and compares the ordinate of the point $DC \cap \{x = x_C\}$ and the one of $AB \cap \{x = x_C\}$.

The heights are 0. for DC and 0.1 for AB: thus DC is below AC in C, which is right, but contradictory with the wrong informations stored in the data structure. Thus the program searches DC in the left subtree of Y, and it cannot find it since it is in the right one. This situation is theoretically impossible and a fatal failure occurs, like say `Nil pointer dereferenced`.

• **If I^* is not forgotten**. Suppose now that the second test is used, or any test such that I^* is found, when the sweeping line passes D. So DC is cut into DI^* and I^*C, AB into AI^* and I^*B. I^* is then inserted in X. Theoretically $I < C$ but for the program: $C = C^* < I^*$. Thus I^* is inserted in X just after $C = C^*$. Thus the next point to be swept is C. Here, the program has to remove the segment I^*C from Y (since C is now the right endpoint of I^*C). But this segment has not yet been inserted in Y. Again, a fatal error occurs.

A possible solution is explained in section 3.4.1.

It is worth comparing the behaviour of the naïve method in $O(n^2)$ which compares all couples of segments, and the one of Bentley and Ottman's method. Obviously, the naïve method is slower (at least when the number of intersection points k is less than $O(n^2)$), but it never crashes; it can provide some wrong results, but the latters are not propagated, contrarily to Bentley and Ottman's method. The naïve method is much more robust against inaccuracy.

2.4 Consequences on a topological method

In $2D$ let P be a set of vertices and $S \in P \times P$ a set of non crossing segments, *ie* two distinct segments can only cut each other at a known common vertex belonging to P. For instance some method has been used to compute all intersection points between an initail set of intersecting segments.

Thus S and P define what is called a *planar map*: a combinatorial structure made of vertices, segments and faces, and supporting various topologic relations of incidency, contiguity or inclusion. Some applications (say Geographic Information Systems) need data structures for modelling such planar maps.

An important notion is the one of *half-edges*. A segment is made of two half-edges, the left and the right. The left half-edge (respectively the right one) contains the left vertex (respectively the right one) of the segment, or in more general words its smaller (respectively its greater) vertex for the lexicographic order. Thus if the segment is vertical, the left (respectively the right) half-edge is the below (respectively the above) one. The two half-edges of the same segment are said to be *complementary* of each other.

Moreover each half-edge e having vertex say v is linked with its *neighbour*: it is an half-edge also incident to v, the first one which is met when turning counterclockwise around v and starting from e. In this way all half-edges sharing the same vertex v are cyclically linked around v, in the counterclockwise orientation. It is obvious that starting from any initial half-edge, and following the neighbour link will always yield to the starting half-edge... when the planar map is correct.

This *neighbour* link makes also possible to follow face contours: Starting from any half-edge e_1, take the complementary of its neighbour to get e_2, and then start again from e_2 to get e_3, and so on until the initial half-edge is reached: this way all half-edges e_1, $e_2 \ldots e_1$ of the same contour are listed. Here again, starting from any half-edge, this travel will always yield to the initial half-edge... when the planar map is correct.

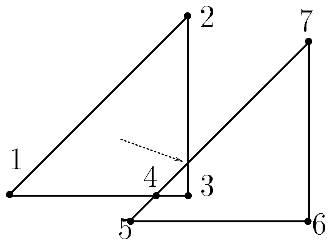

Figure 4: *A vertex has been forgotten: due to an inconsistent topology, following the contours will lead to troubles. Here only one contour will be found, which contains all half-edges: ...(14), (21), (32), (43), (74), (67), (56), (45), (34), (23), (12), (41), (54), (65), (76), (47), (14)..., where (ij) is the half-edge containing the vertex i. Of course the right planar map has one outer contour and three inner ones.*

Now, these two properties cannot be guaranteed when the planar map is wrong, for instance when some intersection points have been forgotten (see Fig. 4), or when half-edges incident in the same vertex are too close to be correctly ordered by the neighbour link when using a *fp*-arithmetic. In fact, the dual methods of following a contour and of turning around a vertex until the starting half-edge is reached, may enter in an infinite loop, because never yielding to the initial half-edge (for instance with a half-edge sequence like: e_1, e_2, e_3, e_4, e_5, e_6, $e_3 \ldots$). The method may also terminate, but provide inconsistant contours, for instance self intersecting ones (see Fig. 4), which will give troubles to say a colouring method, or a method locating points in the planar map.

2.5 Other examples of contradictions

The methods proposed in the Computational Geometry field are theoretically fast because they exploit *properties of consistent geometric objects*, like properties of total orders, algebraic identities, geometric theorems. Maybe the simplest example is the use of the transitivity of the total orders: if it is known that $a < b$ and $b < c$, then it is deduced that $a < c$ without comparing a and c. This property is the basis of the binary search technique used in Bentley and Ottman's method to handle the X and Y data structures. We have seen how inaccuracy ravages this kind of method.

Basically, inaccuracy invalidates geometric and mathematical properties. This section now gives several examples of such properties, often used by geometric algorithms, but ruined by inaccuracy. It would be too lengthy to detail the consequences on geometric algorithms relying on such property, but they would now be obvious after the previous examples.

2.5.1 Example 1

The *power* of a point $M = (x, y)$ relatively to a line D with triple (α, β, γ) is $\alpha x + \beta y + \gamma$. Suppose that A belongs to the line D, because for instance A has been defined as the intersection point between D and another line. Theoretically, this power must be 0. Using *fp*-arithmetic, it is unlikely to be true. So a data structure can store topological informations, like $A \in D$, that will be contradicted by numerical computations.

2.5.2 Example 2

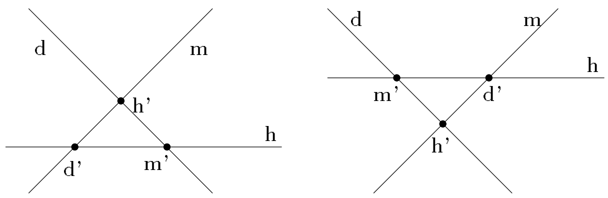

Figure 5: *The two possible lexicographic orders, in the generic case.*

Consider the three lines of figure number 5. h is nearly horizontal, m climbs and has slope about 45 degrees, d goes down and has slope about -45 degrees. Let the three intersection points be $h' = d \cap m$, $m' = d \cap h$, $d' = m \cap h$. If we exclude the degenerate case where all intersection points coincide, then there are only two possible lexicographic orderings for them, either $d' <_L h' <_L m'$ or $m' <_L h' <_L d'$; in the two cases, h' is between m' and d'. Now, if the triangle is small enough, the coordinates of the three points will differ only by their least significant bits, that will be likely corrupted by rounding; so *fp*-arithmetic will sometimes produce non consistent triangles, impossible to draw.

2.5.3 Example 3

The intersection point between a vertical line AB and an oblique one is a point Ω with the same abscissa as A, and B. But the evaluation of the previous formula with *fp*-arithmetic will the more often give a slightly different point Ω^*, such that either $x_A < x_{\Omega^*}$ or $x_{\Omega^*} < x_A$. Actually this also holds for 'almost vertical' line or segment, as previously seen.

This means that relying on the property : $\Omega \in [A, B] \Leftrightarrow A <_L \Omega <_L B$ (to test if Ω really belongs to the segment, not only the straigth line) is doomed to failure.

2.5.4 Example 4

If β and β' are both different from zero, the three following definitions for y_Ω are algebraically equivalent:

$$\frac{\alpha'\gamma-\alpha\gamma'}{\alpha\beta'-\alpha'\beta} \quad , \quad -\frac{\alpha x_\Omega+\gamma}{\beta} \quad \text{and} \quad -\frac{\alpha' x_\Omega+\gamma'}{\beta'}.$$

but evaluated with *fp*-arithmetic, they will generally yield different results. This is quite a serious problem, as programmers frequently rely on such identities to detect the equality between two objects constructed in two different and legal ways.

2.5.5 Example 5

In the projective plane, if three distinct points P_1, P_3, and P_5 are collinear, and if the three distinct points P_2, P_4, and P_6 are also collinear, so are the three points $P_1P_2 \cap P_4P_5$, $P_2P_3 \cap P_5P_6$, and $P_3P_4 \cap P_6P_1$ after Pappus's theorem. Numerical inaccuracy likely prevents the detection of this property. Actually the same holds for all geometric theorems !

2.5.6 The combinatory underlying geometry

This section intends to give an intuitive insight on the combinatoric properties underlying geometric objects. The simplest example is perhaps the one of N points in the plane. For each triple (A, B, C) of distinct points, we define $|ABC|$ to be:

$$|ABC| = sign(\begin{vmatrix} 1 & 1 & 1 \\ x_A & x_B & x_C \\ y_A & y_B & y_C \end{vmatrix})$$

the determinant is twice the signed area of triangle ABC. As well known, either ABC turns on the left and $|ABC| = +1$, or ABC turns on the right and $|ABC| = -1$, or ABC are collinear and $|ABC| = 0$.

There are $3^{\binom{N}{3}}$ ways to assign signs for all $\binom{N}{3}$ triples. But of course very few of them are geometrically possible: triples signs are not independant. For instance they must verify $|ABC| = |BCA| = |CAB|$, and $|ABC| = -|BAC|$. Other less obvious rules are: $|ABC| = |BCD| = |CDA| = |DAB| \Rightarrow |ABD| = |BCA| = |CDB| = |ABC|$ (hint: $ABCD$ is convex), and $|ABP| = |BCP| = |CAP| \Rightarrow |ABC| = |ABP|$ (hint: P is inside the triangle ABC). See [Knu92] for details.

Of course, when the N points are given by their coordinates, it may happen that computing all $|ABC|$ with *fp*-arithmetic yield inconsistent signs.

2.6 Consequences of inaccuracy

To conclude this section about consequences of inaccuracy, inconsistent decisions from numerical imprecision have two kinds of consequences on geometric algorithms.

Either the algorithm crashes or enters in an infinite loop. It is typically the case for the most sophisticated and efficient methods proposed by Computational Geometry, that rely on various geometrical or mathematical properties, like the transitivity of orders, algebraic identities or theorems ... and that propage (sometimes corrupted) intermediate results, for instance: $a < b$ and $b < c \Rightarrow a < c$. These methods enter in theoretically impossible situations: trying to delete an element which has not yet been inserted from some data structure, or ending up in an infinite loop while scanning a theoretically finite sequence.

Or the algorithm terminates normally; it is generally the case for 'brute-force' algorithms, too much stupid to exploit intermediate results and geometric consistency. However these algorithms yield inconsistent results, for instance a graph supposed to be planar will not be (see example in figure 4). Another algorithm, though mathematically correct and taking this result as its input, may crash or infinitely loop or provide inconsistent results in turn.

Practitioners (programmers or users of geometric modellers) were aware of the inaccuracy difficulties since the first geometric modellers, in the seventies; to overcome them, programmers and users have developped empirical tricks, described below. These tricks do not always work; however, people in the CADCAM communauty put up rather willingly with these limitations : first engineers are used with the limitations of the various 'physical' devices they use, and then they consider that, anyhow, it is impossible to do otherwise.

Thus actual commercial geometric modellers can not work without the active complicity and understanding of their users, who have learned to avoid the easiest traps, and who accept to slightly modify their problems until their geometric modeller works... In the CADCAM field, this is called the *robustness problem*.

Among theorists of *Computational Geometry*, the awareness of the inaccuracy problem is much more recent, say the late eighties, with Milenkovic's work [Mil88]: Computational Geometry assumes a theoretic model for computers, where each arithmetical operation is performed exactly in constant time and space. When known, the inaccuracy problem was not considered as worthy of research, it was just a concern of programming. This is probably the reason why there is up to now no available library solving problems of *Computational Geometry*.

3 Fighting against inaccuracy

This section surveys proposed approches to solve or bypass the inaccuracy problem.

3.1 The ϵ heurism and other probabilistic approachs

3.1.1 The popular ϵ heurism

To overcome inaccuracy, the most popular trick used in geometric modellers is the ϵ heurism. When two *fp*-numbers differ by less than a given threshold traditionally called ϵ, they are considered to be the same. The test may be made in an absolute manner : $|a-b| < \epsilon$, or in a relative one : $|a-b| < \epsilon \times max(|a|, |b|)$. Some modellers use several ϵ, say one for lengths, another for areas, another for angles.

This heurism lost the transitivity of the equality : it is easy to find a, b and c such that $a =_\epsilon b$, $b =_\epsilon c$, but $a \neq_\epsilon c$ where $=_\epsilon$ means 'equal for the ϵ heurism': thus inconsistencies remain possible.

Moreover, finding the relevant value(s) for ϵ(s) is a very difficult task, depending on the usual range of numbers (it depends on the applications), and on the format of used *fp*-numbers : it is common folklore in the CADCAM communauty that the conversion from 32-bits *fp*-numbers to 64-bits has needed a not so easy updating of ϵs. Of course the ϵ heurism may fail, and it does sometimes. In practice, it seems to work not so bad and to improve the robustness of the geometric modellers, but it is also due to the active complicity of their users!

3.1.2 Gap arithmetics

The ϵ heurism is based on a right intuition, following Canny's gap theorem [Can88]:

Canny's gap theorem : *Let x_1, $x_2 \ldots x_n$ be the solutions of an algebraic system of n equations and n unknowns, having a finite number of solutions, with maximal total degree d, with relative integers coefficient smaller or equal to M in absolute value. Then, for all $i \in [1, n]$, either $x_i = 0$ or $|x_i| > \epsilon_c$ where*

$$\epsilon_c = \frac{1}{(3Md)^{nd^n}}$$

This theorem gives a way to *numerically* prove that a number is null : compute a (guaranteed) interval containing it, with width smaller than ϵ_c. As soon as the interval does not contain 0, the number is clearly not 0 and its sign is known. Otherwise, if the interval contains 0 and has width less than ϵ_c, the number can only be 0.

Alas, there are several problems. First ϵ_c is much, much smaller than the epsilon used in geometric modellers; actually ϵ_c is generally much smaller than the smallest positive *fp*-number, even in simple examples. So, an extended arithmetic providing big floats is needed. Secondly, even if such an arithmetic is available, such a computational scheme will have an exponential cost : an exponential number of digits is needed to prove the nullity of a number, because of the term nd^n in Canny's theorem. Now, there is no hope to significantly improve the Canny's gap in the worst case, because it is already sharp; it is almost reached in the following simple case : $x_1(Mx_1 - 1) = 0$, $Mx_2 - x_1^2 = 0$

$\ldots Mx_n - x_{n-1}^2 = 0$. A possibility will be to find a more convenient ϵ depending on the system at hand, and not only on d, M and n. I am currently not aware of people using this kind of techniques.

A variant of this gap arithmetic is possible in the rational case, *ie* when only rational numbers and operations are used. The idea is to maintain for each met number x an upper bound of the number of digits of its denominator: $d(x)$, and another bound for the number of digits of its numerator: $n(x)$, for a given base, say $B = 2$. These upper bounds are known for the input numbers, and they are computed as follow for $x + y$, xy, $1/x$ and $-x$, without explicitly computing the exact rational form:

$$\begin{array}{l} n(x+y) = 1 + max(n(x) + d(y), n(y) + d(x)),\ d(x+y) = d(x) + d(y) \\ n(x \times y) = n(x) + n(y),\ d(x \times y) = d(x) + d(y) \\ n(1/x) = d(x),\ d(1/x) = n(x) \\ n(-x) = n(x),\ d(-x) = d(x) \end{array}$$

Now, to know if a number x vanishes, it is enough to compute a good enough approximation x^* of x, by using some on-line bigfloat library: the smallest (in absolute value) non-zero rational number, having denominator with at most $d(x)$ digits in base B, is $\pm\epsilon_x$ with $\epsilon_x = \frac{1}{B^{d(x)}-1}$. Thus x vanishes iff $x \in]-\epsilon_x, \epsilon_x[$, or more conveniently when $x \in [-B^{-d(x)}, B^{-d(x)}]$. When a number appears to be 0, its fields n and d, and the ones of its dependent numbers may be strengthened on the fly.

The previous scheme may be extended for other fields than $\mathbb{Q}$, for instance quadratic algebraic extensions $\mathbb{Q}[\sqrt{a}]$, where $a > 0$ is not a square in $\mathbb{Q}$. The main idea is to maintain an upper bound of the size of an *exact* and *virtual* representation, *virtual* meaning that this exact representation is never computed. From this bound for any number z, it must be possible to effectively deduce a gap ϵ_z such that $z \in]-\epsilon_z, \epsilon_z[\Rightarrow z = 0$. In the previous example $\mathbb{Q}[\sqrt{a}]$, the smallest non vanishing number $z = x + y\sqrt{a}$, $x, y \in \mathbb{Q}$, with an upper bound on the size of x and y, is such that $-\frac{x}{y}$ is a convergent to $\sqrt{a}$ in the continued fraction expansion of $\sqrt{a}$, and the classic theory of continued fraction expansions of quadratic numbers [Bak84] provides an explicit ϵ_z gap.

3.1.3 Other heurisms

Another more practicable but only probabilistic method is to compute with some big floats library, and to use the ϵ heurism, with say 10^{-200}, and hope that life will not be so bad to provide a counterexample... Some people currently investigate such an approach, but I don't know any publication.

Another probabilistic trick stems from modular arithmetic. The idea is to perform all computations modulo one (or several) finite field $\mathbb{F}_n$ (for instance $\mathbb{Z}/n\mathbb{Z}$ where n is a prime integer, about say 2×10^9). Clearly, if $a \bmod \mathbb{F}_n$ does not vanish, a cannot be 0, even when a^* is very small: we have to precise a^* in some way to reliably find its sign, for instance using some bigfloat library or some on-line arithmetic. On the other hand, if a^*

is small, and if $a \bmod \mathbb{F}_n$ vanishes, one can take the risk to assume $a = 0$. This scheme is straightforward to implement when only rational operations ($+$, $-$, $\times$, $:$) and numbers are used, because each rational number has only one homomorphic element in the finite field. The only difficulty arises when a division by 0 occurs in the finite field, which is very unlikely. Such a scheme has been investigated by A. Agrawal and A. Requicha [AA94], and by M. Benouamer and his colleagues [BJMM94] (see below section 3.4.3 about hash coding lazy numbers). However this approach becomes more problematic when algebraic non rational operations and numbers are involved: for instance each quadratic number have two homomorphic images in the finite field (or in its cloture) and it is impossible to discriminate them, for example to distinguish a positive and a negative square root in $\mathbb{F}_n$. Thus to know if an algebraic number vanishes, all its homomorphic images in $\mathbb{F}_n$ or its cloture must be tested. As far as I know, this scheme has never be investigated in the algebraic non rational case, for geometric computations.

3.2 Careful programming

Some computer scientists prefer to avoid the ϵ heurism and have settled a set of tricks, say :

• Check first in the data structures before computing; for instance, before computing the power of a vertex relatively to some line, verify first if the line is topologically incident to the vertex from the data structures at hand. This avoid the inconsistency 2.5.1.

• Handle in a special way some particular cases, for instance the intersection point between a vertical line and an oblique one : in this case, assign the abscissa with the abscissa of the vertical line, not with the expression $\Delta x/\Delta$. This avoid the inconsistency 2.5.3.

• Do not use several distinct formulas for the same value (though it seems to contradict the previous rule...). This avoid the inconsistency 2.5.4.

• The computation of $ab \cap cd$, $ab \cap dc$, $ba \cap cd$, $ba \cap dc$ (they are, say, segments in $2D$) generally give slightly different results. Before computing an intersection, it is worth systematically orienting segments at hand, so that $a <_L b$ and $c <_L d$ and so on, by exchanging vertices, so that we will indeed have : $ab \cap cd = ab \cap dc = ba \cap cd = ba \cap dc$.

M. Iri and K. Sugihara [IS89] used this kind of approach for computing Voronoï's diagrams. They ensure that their program will never crash because of inaccuracy, that the resulting graph is correct when there is no numerical difficulty, and otherwise that the graph is connected with all vertices having degree 3, like a correct Voronoï's diagram. It is impressive, but there is no guaranty that another program, mathematically correct, and using this Voronoï's diagram as an input, will not crash !

To conclude, all these stratagems are sometimes clever but they can only avoid the more obvious inconsistencies. But avoiding more convoluted ones (for instance the non respect of Pappus's theorem, see 2.5.5) seems an impossible task: the next section explains why.

3.3 Respecting Consistency: the Quest

The aim of careful programming, of Iri and Sugihara's work [IS89], and of some V. Milenkovic's work [Mil88], can be paraphrased as:

In Numerical Analysis, it is possible to find an approximation of the solution of the problem at hand which is the exact solution of another but close problem of the same kind. It is even possible to measure in some way these distances. We would like to do the same in geometric algorithms, *ie* to find an approximation of the solution that is the exact solution of another close problem.

It means that, at least, the found solution must be 'geometrically realizable'. To take an example, let us turn back to the problem in section 2.5.6: let $P_1, P_2 \dots P_n$ be n points in the plane. We want to compute with *fp*-arithmetic any of the $|ABC|$; we accept that some triple-sign to be wrong (relatively to exact arithmetic), but we want to get a consistent signs set, which is the signs set of another set of points $Q_1, Q_2 \dots Q_n$, close to $P_1, P_2 \dots P_n$.

This algorithm will be something like this: all triple signs ABC (yes, there are $O(n^3)$ such triples, but here our problem is not to obtain an efficient algorithm, but a robust one!) are computed straigthforwardly with *fp*-arithmetic, by the devoted formula, and with an error bound. The value of the majority of signs will be clearly positive, or clearly negative. For remaining ambiguous signs, we want to resort to some oracle who will give us a set of missing signs that is geometrically realizable.

The question now becomes: is it possible to decide if a partial signs system implies such other missing sign, or equivalently, if a given sign system is consistent (geometrically realizable) or not?

This problem is not only a 'toy problem', because a lot of methods (convex hull computations, intersection between polygons, for example) proposed by Computational Geometry in $2D$ can be reformulated so that they will *only* use triple-sign in geometric tests. For instance, two segments pq and rs intersect each others (with no degeneracy) iff $|pqr| \times |pqs| = -1$ and $|rsp| \times |rsq| = -1$. Thus, for applications where all geometric tests can be reformulated only with triple-sign test, the robustness problem will be theoretically solved. It is worth remarking that this approach can be extended to $3D$ and beyond.

The combinatoric properties of the sign systems of triples of points in $2D$ have been studied. They are partly characterized, notably by D. Knuth's CC (*CounterClockwise*) axioms (see Fig. 6), or equivalently by uniform acyclic oriented matroids of rank 3: see [Knu92] for details. Though illuminating, this axiomatic system and the corresponding combinatorial structure are too poor to capture all properties of the 'true' geometry: for instance, Pappus's theorem is not a consequence of Knuth's axioms, and one can find some sign systems that though verifying Knuth's axioms are not geometrically realizable.

Up to now a complete combinatoric characterization is not known, and perhaps there can be no finite set of purely combinatoric axioms (in Knuth's style, *ie* not using continuity, or coordinalization by some continuous fields like $\mathbb{C}$ or $\mathbb{R}$ and so on) that characterizes

Axiom 1 (cyclic symmetry). $pqr \Rightarrow qrp$
Axiom 2 (antisymmetry). $pqr \Rightarrow \neg prq$
Axiom 3 (nondegeneracy). $pqr \vee prq$
Axiom 4 (interiority). $tqr \wedge ptr \wedge pqt \Rightarrow pqr$
Axiom 5 (transitivity). $tsp \wedge tsq \wedge tsr \wedge tpq \wedge tqr \Rightarrow tpr$

Figure 6: *The five Knuth's axioms, to explore the combinatoric properties of the generic sign systems of triple of points. pqr means:* $|pqr| = 1$ *and* $\neg pqr$ *means* $|pqr| = -1$.

geometrically realizable systems: see [Knu92] pp 96 for more. Another very bad new is that the decision problem (is this given set of triple-signs consistent?) for Knuth's sign systems is NP-complete.

Actually, it seems that the problem is even more complicated. D. Knuth only considers generic situations, where the triples have sign $+1$ or -1, but never 0. Now some degenerate configurations are realizable in some fields (say the algebraic real field) but not in others, like $\mathbb{Q}$. Such a configuration is given in B. Grünbaum's book [Grü67] and illustrated in Fig. 7: Place nine points A, B, C, D, E, F, G, H, I so that the following subsets of points, and only these subsets, are colinear: $ABEF$, ADG, AHI, BCH, BGI, CFI, DEI, DFH, and obviously of couples of points. There are 2 solutions, needing regular pentagons, thus this configuration is not realizable in $\mathbb{Q}^2$, but it is in $\mathbb{R}^2$, actually in $\mathbb{Q}[\sqrt{5}]^2$.

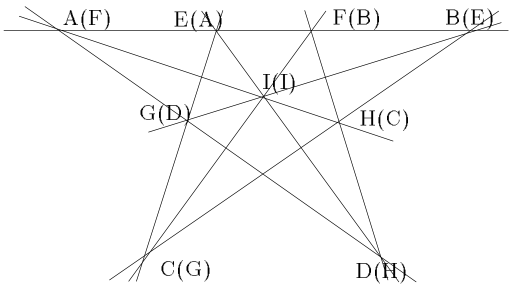

Figure 7: *This above configuration is not realizable in* $\mathbb{Q}^2$*, but it is in* $\mathbb{R}^2$.

In conclusion, the approach of 'Computing with *fp*-arithmetic but respecting geometric consistency' is not so easy to achieve.

3.4 Exact arithmetics

An obvious solution against inaccuracy is the use an exact arithmetic. Alas, even when an exact arithmetic on big integers or big rational is sufficient, a straigthforward implementation is much too slow. From an experiment of M. Karasick, D. Lieber and L.R. Nackmann [KLN91], the Voronoï's triangulation of 10 random points in $2D$ needed 0.1

second; that of 10 random points with rational coordinates (2 digits for the numerator, 3 for the denominator, with radix 2^{16}) needed 1200 seconds with a standard rational library and generated intermediate values up to 81 digits long. Of course computers are now faster, but the order of magnitude of the ratio between the two running times is still correct. It is easy to understand why exact arithmetic are so seldom used in geometric modellers.

In some cases, the machine numbers are sufficient to achieve exact computations, with some tricks. Section 3.4.1 presents this fast but limited solution.

When this solution does not apply, one can envisage to exploit the fact that the *fp*-arithmetic (or some interval arithmetic to have an upper bound of errors) is very often sufficient to decide the sign of an expression, and to use an exact arithmetic only when the *fp*-arithmetic is not reliable. In practice, this idea is implemented in several way, this paper only presents the *LN* library due to S. Fortune and C. Van Wyk [FVW93] in section 3.4.2, and the lazy exact arithmetic due to M. O. Benouamer, P. Jaillon, J-M. Moreau and the author [BJMM93, MM96] in section 3.4.3. Due to lack of space, other approaches in the same tendency cannot be detailed but are worth mentionning: [OTC87, KLN91, Yam87, GT91, NSTY93].

3.4.1 The poor man's exact arithmetic

In some restricted cases, it is possible to use an exact arithmetic which is as fast as the *fp* one. This section describes the various tricks I have used in 1982–1983 to implement such an arithmetic, for a $2D$ graphic editor [GM84, Mic87, GHPT89]: Probably other people confronted with inaccuracy problems have used similar tricks at this time, but very few of them were published –if any– because the latters were considered shameful in the Computational Geometry field and inaccuracy was not considered as a relevant issue at this time.

The $2D$ graphic editor used the Bentley and Ottman's method to compute the intersection points between the data segments. First the coordinates of the initial vertices were rounded on integers in the range 0 to $G = 30,000$: it was not a trouble for the application. Thus the straight lines equations could also be stored in 3 `int` (machine integer): (α, β, γ) such that $|\alpha| \leq G$, $|\beta| \leq G$ and $|\gamma| \leq 2G^2$, a trivial consequence of formula 1 in section 2.1. The intersection points between segments ($x = \frac{\Delta x}{\Delta}, y = \frac{\Delta y}{\Delta}$) could be represented (assuming w.l.o.g. that $\Delta > 0$) by an `int` tuple ($x_e = \lfloor\frac{\Delta x}{\Delta}\rfloor, x_r = \Delta x \bmod \Delta, y_e = \lfloor\frac{\Delta y}{\Delta}\rfloor, y_r = \Delta y \bmod \Delta, \Delta$): it is easy to see that $0 \leq x_e, y_e \leq G$, and that $0 \leq x_r, y_r < \Delta \leq 2G^2$. Some temporarily needed values, such Δx or Δy, could exceed the maximum `int` value, but these computations were exactly performed using `double` numbers. A final trick was used to compare and sort these coordinates: the comparision of the two rational numbers $\frac{a}{b}$ and $\frac{c}{d}$, with $0 \leq a < b$ and $0 \leq c < d$, cannot reduce to the comparision of ac and bd, since in some cases, these values were too large to be exactly represented by `int`, or even by the mantissa of `double` numbers. Using a simultaneous continuous fraction expansion of $\frac{a}{b}$ and $\frac{c}{d}$, it is possible to say that ([Mic87] pp 38):

$$\mathrm{order}(\frac{a}{b},\frac{c}{d}) = \mathrm{order}(\frac{d}{c},\frac{b}{a}) = \mathrm{order}(\lfloor\frac{d}{c}\rfloor + \frac{d \bmod c}{c}, \lfloor\frac{b}{a}\rfloor + \frac{b \bmod a}{a})$$

If $\lfloor\frac{d}{c}\rfloor \neq \lfloor\frac{b}{a}\rfloor$, then:

$$\mathrm{order}(\frac{a}{b},\frac{c}{d}) = \mathrm{order}(\lfloor\frac{d}{c}\rfloor, \lfloor\frac{b}{a}\rfloor)$$

otherwise:

$$\mathrm{order}(\frac{a}{b},\frac{c}{d}) = \mathrm{order}(\frac{d \bmod c}{c}, \frac{b \bmod a}{a})$$

Since $c < d$ and $a < b$, the recursion eventually terminates. For instance,

$$\begin{aligned} \mathrm{order}(\frac{2}{7},\frac{3}{10}) &= \mathrm{order}(\frac{10}{3},\frac{7}{2}) = \mathrm{order}(3+\frac{1}{3}, 3+\frac{1}{2}) = \mathrm{order}(\frac{1}{3},\frac{1}{2}) \\ &= \mathrm{order}(\frac{2}{1},\frac{3}{1}) = \mathrm{order}(2+\frac{0}{1}, 3+\frac{0}{1}) = \mathrm{order}(2,3) = \mathrm{smaller} \end{aligned}$$

Remark: Obviously, the same trick can be used to compute the sign of the determinant $\begin{vmatrix} a & c \\ b & d \end{vmatrix}$: this idea has since be used and extended to 3 by 3 determinants with integer entries [ABD+95] by F. Avnaim, J-D. Boissonnat, O. Devillers, F.P. Preparata and M. Yvinec.

Despite its interests, the limitations of such tricks are obvious. It cannot work in $3D$ or beyond because the computation depth increases, so involved numbers become too big to be exactly representable by machine numbers. For the same reason, algorithms cannot be reentrant.

3.4.2 The LN library

S. Fortune and C. van Wyk proceed in two stages: First the program is pre-compiled and the minimum number of digits needed for the exact arithmetic (the longest integer generated by the algorithm, knowing the data range and the arithmetic expressions in the program) is determined. For each test in the program, they automatically generate *C++* code:

1. to compute the test in standard *fp*-arithmetic, using references to original data only;
2. to test if the *fp*-value is greater than the maximum possible error for the expression;
3. finally, to call the exact, long integer library to evaluate the expression.

The program is then compiled and linked with the exact library. Note that every test must be made with reference to original data. This permits a static (*ie* before running-time) computation of the maximum possible error for each expression when evaluated in *fp*-arithmetic; so the error bound has not to be computed at run time with intervals or whatever method. It speeds up execution, but it is not always very convenient for the user [CM93]; it forbids on-line and reentrant algorithms, where the depth of computation is not *a priori* known.

3.4.3 The lazy arithmetic

The lazy arithmetic computes with lazy rational numbers. A lazy rational number is first represented by an interval of two *fp*-numbers, guaranteed to bracket the rational number, be it known (exactly evaluated) or not; and then by a symbolic definition, to permit recovering the exact value of the underlying rational number, if needed. The definition is either a standard representation of a rational number (for example 2 arrays or lists of digits in some basis, for the numerator and the denominator), or the sum or the product of two other lazy numbers, or the reciprocal or the opposite of another lazy number. Thus each lazy number is the root of a tree, whose nodes are binary (sum or product) or unary (opposite or reciprocal) operators, and whose leaves are usual rational numbers; actually, lazy numbers form a directed acyclic graph rather than a tree, since any node or leaf may be shared. Each operation is generally performed in constant time and space : a new cell is allocated for the number, its interval is computed from the intervals of the operand(s), and the definition field is filled (operation type, and pointers to the operand(s)). Intervals are the more often sufficient during computations; the only cases when they become unsufficient, and when the definition has to be 'evaluated' (*ie* with rational arithmetic) are : when one wants to compare two lazy numbers the intervals of which overlap, when one wants the sign or the reciprocal of a lazy number whose interval contains 0. The evaluation method is the natural and recursive one. To summarize, rational computations are postponed until they become either unavoidable : they are done, or useless in the majority of the cases : thus they will never be done if they are useless. Using such a lazy library is transparent: classical geometric methods need not to be modified.

The lazy library also provides hashing of lazy numbers. Hashing techniques typically permit to recover topologic data from numerical ones, for instance vertices from coordinates. Obviously this technique needs to compute hash codes from numbers. Here we face a difficulty since the exact value of lazy numbers is unknown, and approximations are not relevant for reliably computing hash keys. The solution stems from modular arithmetic [BJMM94, MM96].

Contrarily to LN, the lazy library is fully dynamic and so equally applies to on-line and reentrant algorithms : the computation depth needs not to be known *a priori*. In compensation, LN when usable should be a little faster than the lazy library.

3.4.4 Open problems

These two arithmetics have serious limitations : they apply only when a rational arithmetic is sufficient; but geometric problems met in the real world involve for instance intersection between algebraic curves or surfaces. Rotations by $k\pi$ with $k \in \mathbb{Q}$ also introduce algebraic numbers. Or lengths of some squares…it is an old story ! For the moment, no lazy algebraic arithmetic has been proposed.

3.5 Interval Confining

The ϵ heurism lost the transitivity of ordering (it is possible to have $a =_\epsilon b$, $b =_\epsilon c$ and $a \neq_\epsilon c$), so inconsistencies remain possible. In such a case, a solution is to abandon the distinction between a, b and c, and to merge them into another larger entity, actually an interval. Computations are performed with an interval arithmetic or another equivalent method providing error bounds; as soon as two entities overlap, they are merged in a third larger entity that contains the two previous ones. One can remark that two close but non overlapping entities have to be merged when is introduced a third entity that overlap the two first ones: one may deplore this loss of information (the distinction between the first two entities is lost, whereas they are not modified), but it is the spirit of this approach, the principle that ensures its consistency. The example of figure 5 will become something like figure 8. This approach has been investigated by M. Segal [Seg90] and by D. Jackson

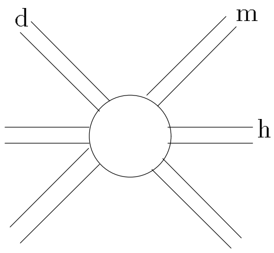

Figure 8: *Three lines with their halos, incident to a fuzzy point (the circle).*

[Jac95] in solid modelling. In $3D$, geometric elements (vertices, edges or arcs, surfaces) are surrounded by a thin halo of imprecision; two distinct and not adjacent elements must have not overlapping halos. During say the computation of some boolean set operation (intersection or union or difference between two 'solid' geometric objects), two elements the halos of which are found to overlap must be cut or merged to restore the consistency of the data structures. In 1995, D. Jackson has implemented this way a robust algorithm to compute boolean set operations between $3D$ geometric objects with curved surfaces.

The main advantages of this approach are that it applies not only to 'linear' problems but also to algebraic ones, and that it does not rely on an exact arithmetic; so it is fast. Moreover, it is intuitive. Finally, it can handle in a natural way inaccurate data: either these data are obtained from some sensors and thus are known only up to some precision, or on the other hand, the modelling stage has taken into account the fact that mechanical objects can be manufactured only within some tolerance. Up to now, it is the only approach that can represent fuzzy data.

Its drawbacks are that all algorithms must be modified. Moreover it is also not clear for the moment that this approach really solves the inaccuracy issue. For instance, the distance between two geometric elements can be computed in several but algebraically equivalent ways; with a first formula, and in *fp*-arithmetic, one may find that two elements do not overlap, but they will with another formula: *so contradictions remain possible.*

3.6 Bypassing the inaccuracy problem

3.6.1 The *CSG* representation

You can always try to bypass problems you do not know how to solve: in the CADCAM communauty (by opposition to the more theoretician communauty of Computational Geometry), this tendency emerges since the conference CSG94 and CSG96 [csg94]: It considers that algorithms or data structures that do not withstand inaccuracy are say paranoiac and must be avoided. First, this tendency wants to get rid of methods from Computational Geometry: to achieve a good complexity, these last methods rely on geometric consistencies and are made much more sensitive to inaccuracy than 'brute force' methods. Second, this tendancy rejects topology based data structures, like the Boundary Representations (BRep for short). In a nutshell, BReps explicitly handles representations for vertices, edges and surface patches, and all the topologic incidency relations between them; they are very explicit but their redundancy (does this vertex numerically lies on this surface though it topologically does?) exposes them to inconsistencies. In particular, it is known that robustness is exceedingly difficult to achieve when performing boolean set operations between geometric objects represented by Breps. The new tendency prefers CSG representations.

CSG representations (*Constructive Solid Geometry*) describe objects in only an implicit way, by CSG trees. A leaf of a CSG tree carries a *primitive* object, described by some (typically algebraic) inequation $f(x, y, z) < 0$, for instance a quadric or a torus. A node is either the union or the intersection or the difference between other CSG trees. Mathematically speaking, a CSG is a semi algebraic set, modulo some regularization problems (is it $f(x, y, z) < 0$ or $f(x, y, z) \leq 0$?) which are not relevant here.

Thus the contour of the object represented by a CSG tree is not explicitly described, and it is not obvious that a CSG tree does not describe only the empty set, contrarily to BReps. But there exist very robust methods to display objects defined by CSG trees, say by ray-casting methods, and to approximately triangulate them by the so-called marching-methods. We briefly present the principles of these techniques in the following sections, to show they are insensitive to inaccuracy.

For people standing up for the *CSG* approach, the latter solves all problems, not only the inaccuracy one, but also say difficulties met when blending surfaces with Breps. On the other hand, all commercial CADCAM softwares rely on BReps, and perhaps not only by chance ! It is too early to conclude.

3.6.2 Ray-casting methods

Pictures are described in Computer Graphics by $2D$ arrays of points, the so called 'pixels', a shortcut for 'picture element'. To compute such a picture of an object described by a *CSG* tree, the ray-casting method computes which object is seen in each pixel: the eye location and the point to be computed define a half straight line: the ray, whose intersection with the scene has to be computed. When the object is a primitive $f(x, y, z) < 0$, where f is typically a polynomial in x, y, z, this problem boils down to the resolution of an algebraic equation in t: just replace x, y, z in $f(x, y, z) = 0$ by $x = x_e + at$, $y = y_e + bt$, $z = z_e + ct$, where (x_e, y_e, z_e) is the eye location and (a, b, c) the support vector of the ray. The numeric resolution yields the intersection, a set of intervals $[t_0, t_1]$, $[t_2, t_3]$ ...along the ray, where $0 \leq t_0 \leq t_1 \leq t_2 \leq t_3 \ldots$. When the object is a boolean combination, say $A \cap B$ for instance, it suffices to recursively compute the intersection of the ray with subtrees A and B, which give two resulting sets of intervals $A_{[]}$ and $B_{[]}$, and then to calculate $A_{[]} \cap B_{[]}$: a trivial merge.

It is rather easy to protect this method against inaccuracy, since, if a difficulty occurs, it is always possible to cheat and slightly perturb the ray: after all, each pixel stands for a little square area in the picture and not only a point! In the worst cases, some intersection intervals $[t_i, t_{i+1}]$ between the ray and an object $F(t) < 0$ may be forgotten, or added, but this only occurs in two cases: first for exceedingly thin objects which only occur when I decided to make fail the software (see Fig. 9) and secondly, when the ray is almost tangent to the surface of the object: this last situation often happens but it is immaterial for the final picture. Moreover, even when such errors are made, they are not propagated to the other pixels, and the program never crashes.

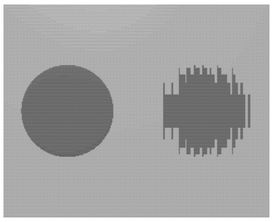

Figure 9: Two ray-traced ellipsoids, with radius 1, and thickness 10^{-5} for the left one, 10^{-7} for the right one. The latter is so thin that some intersections are missed.

3.6.3 Marching-methods

To approximately triangulate objects defined by CSG trees within a given tolerance μ (see [PA94] in [csg94]), the space $\mathbb{R}^3$ is first partitioned with a regular cubic lattice, with side μ; each cube is then partitioned into tetrahedra; for all vertices $v = (x, y, z)$ of the lattice, the value of the CSG tree at v is computed: for a primitive described by an inequation $f(x, y, z) < 0$, it is $f(v)$; for nodes $A \cap B$ and $A \cup B$, it is respectively $max(A(v), B(v))$ and $min(A(v), B(v))$ where $A(v)$ and $B(v)$ recursively stand for the value of CSG trees A

and B at the point v. The surface of the object cut a given tetrahedron when the values at the 4 vertices have opposite signs. These 4 values define, by linear interpolation, a unique linear map $l(x, y, z)$ from $\mathbb{R}^3$ to $\mathbb{R}$, and the plane $l(x, y, z) = 0$ is considered as a good enough approximation of the contour of the object inside the tetrahedron: it gives a triangle or a quadrilateral. The same is done for all tetrahedra. This technique is illustrated in $2D$ in Fig. 10.

Marching-methods are not sensitive to inaccuracy: in the worst cases, a vertex value is close to 0, and *fp* evaluations may yield a wrong sign for the value, but the only and immaterial consequence will be to move a little the approximation surface.

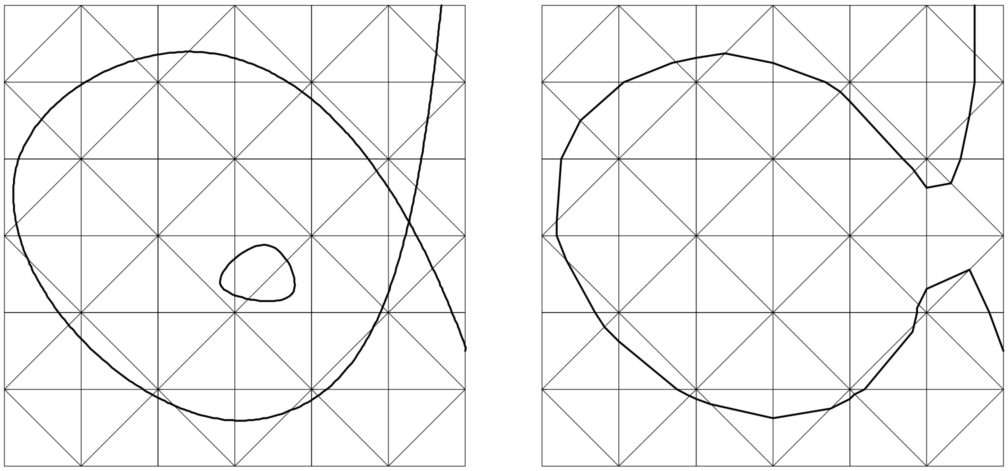

Figure 10: *A $2D$ curve and its piecewise linear approximation. The topology may be different, and some small components of the real curve may be forgotten. But this technique is perfectly robust.*

Of course, it is better to use some optimizations to not consider all cells of the lattice, like some computation by intervals [dFS95, Tau93], or like exploiting the continuity: once a starting tetrahedron crossed by the surface is known, the sides by which the contour surface leaves the tetrahedron are easily computed and the contour surface is then followed in the neighbouring tetrahedron. These optimizations are beyond the scope of this paper, but the reliability of the marching-methods is preserved.

Thus a BRep (and all its precious informations) can be obtained from a CSG tree, without having to perform boolean set operations on BReps, a very unreliable process. But it is possible to go farther and to question the need for a BRep: why not stop at the discretization stage?, as the next section argues.

3.6.4 Discretization

Boundary representations are basically used to approximately 'evaluate' a *CSG* object. However they are not the only possible way, only the usual one, due to the history of the CADCAM field. Discretization is another solution: the space is represented by a $3D$ array of points, the so called 'voxels', a shortcut for 'volume element'. This discrete representation makes trivial the most frequent geometric problems (estimating mass properties, interference detection, boolean operation, etc) and it virtually removes the inaccuracy problem.

Today, Computer Tomography and Magnetic Resonance Imaging make it possible to acquire such image data in $3D$. On the other side, from such a voxel-based representation, Rapid Prototyping [SBE95] can produce real tactile plastic prototypes for manufacturers, chemists or biologists by 'printing in $3D$', using stereolithography: the stereolithography apparatus builts the prototype slice by slice, by laying down a thin layer (between 0.1 and 0.5 millimeters) of liquid resin on the previous slice, instantly curing it into solid plastic, and starting again. Moreover, at this level of precision, almost the molecular level, the voxel-based representation is also the most precise one: this is in contrast with the not so old reluctancy of some theorists for this discrete representation, which they considered as a trivial and very rough approximation of 'exact' CSG models. Last, the voxel-based representation is always the simplest one, obviously.

It is worth comparing the history of the representation of the space with the one of pictures: In the beginning of Computer Graphics and CADCAM, more than twenty years ago, pictures were usually not represented by discrete representations, *ie* $2D$ arrays of pixels, but by boundary representations, because discrete representations were too cumbersome at this time, and available devices only provide wire frame display, which boundary representations were best suited for. The related algorithms, for removing hidden parts for instance, already had troubles with inaccuracy. Nowaday, pictures are represented by discrete representations, and everybody has forgotten these algorithms and their inaccuracy problems. One can wonder if, similarly, the time is not comed for discrete representations of space to supplant boundary representations of solids, and to remove the inaccuracy problem in geometric computations?

4 Conclusion

This paper has shown how crucial for geometric computations the inaccuracy issue is. Some examples has shown the specificity of geometric computations, the fact that below geometry lay deeper combinatorial structures, the non respect of which lead to topological inconsistencies and running time crashs.

This paper has surveyed the more typical proposed approaches to overcome the inaccuracy problem: the arithmetical approach tries to improve the used arithmetic to save geometric methods and also Computational Geometry itself from inaccuracy. On the other hand, the Computer Graphics and CADCAM communauties reject data structure and algorithms considered not robust enough against inaccuracy, typically the boundary representations and theoretical algorithms from Computational Geometry. They argue that simpler algorithms, like ray-tracing or marching-methods, and data structures, like *CSG* trees, ray-representations or voxel-based representations, are insensitive to inaccuracy and potentially solve all problems met by geometric modellers.

It is sure that arithmetic issues are the current crucial challenge for the Computational Geometry field: as long as the inaccuracy problem will not be solved, Computational Geometry will not apply to problems in the real world, and its algorithms and data structures will not be used.

On the other hand, it is not sure that the approach advocated in the Computer Graphics and CADCAM fields definitively avoid the inaccuracy problem: maybe the latter is only defered for a moment, but will soon reappear.

References

[AA94] Agrawal A. and Requicha A.G. A paradigm for the robust design of algorithms for geometric modeling. *Computer Graphics Forum (EUROGRAPHICS'94)*, 13(3):C–33–C–44, 1994.

[ABD+95] F. Avnaim, J-D. Boissonnat, O. Devillers, F.P. Preparata, and M. Yvinec. Evaluation of a new method to compute signs of determinants. In *Proceedings of the 11th Symposium on Compututational Geometry*, pages C16–C17. ACM Press, 1995.

[Bak84] A. Baker. *A concise introduction to the theory of numbers.* Cambridge University Press, 1984.

[BJMM93] M.O. Benouamer, P. Jaillon, D. Michelucci, and J-M. Moreau. A lazy arithmetic library. In *Proceedings of the IEEE 11th Symposium on Computer Arithmetic*, Windsor, Ontario, June 30-July 2, 1993.

[BJMM94] M.O Benouamer, P. Jaillon, D. Michelucci, and J.M. Moreau. Hashing lazy numbers. *Computing*, 53(3–4):205–217, 1994.

[Can88] J. Canny. *The complexity of robot motion planning.* M.I.T. Press, Cambridge, Mass., 1988.

[CM93] J.D. Chang and V. Milenkovic. An experiment using ln for exact geometry computations. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 67–72, Waterloo, Canada, August 5-9, 1993.

[csg94] *Set Theoretic Solid Modelling Techniques and Applications.* Information Geometers Ltd, 47 Stockers Avenue, Winchester, SO22 5LB, UK, 1994. Proceedings of the CSG 94 Conference, Winchester, UK, 13-15 april 1994.

[dFS95] L.H. de Figueiredo and J. Stolfi. Adaptive enumeration of implicit surfaces with affine arithmetic. In *Proceedings Eurographics Workshop on Implicit Surfaces*, pages 161–170. INRIA, 1995.

[EC92] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *Proc. 8th ACM Symp. on Comp. Geometry*, pages 74–82, Berlin, Germany, 1992.

[EM90] H. Edelsbrunner and E.P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph*, 9:66–104, 1990.

[FVW93] S. Fortune and C. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proceedings of the 9th ACM Symposium on Computational Geometry*, pages 163–172, San Diego, May 1993.

[GHPT89] M. Gangnet, J.C. Hervé, T. Pudet, and J.M. Van Thong. Incremental computation of planar maps. *ACM Computer Graphics (SIGGRAPH 89)*, 23(3):345–354, July 1989.

[GM84] M. Gangnet and D. Michelucci. Un outil graphique interactif. In *Proceedings of MICAD 84*, pages 95–110. Hermès, Feb.-Mar 1984.

[Grü67] B. Grünbaum. *Convex polytopes.* London Interscience, 1967.

[GT91] M. Gangnet and J.M. Van Thong. Robust boolean operations on 2d paths. In *Proceedings of COMPUGRAPHICS91*, volume 2, pages 434–443, Sesimbra, Portugal, 1991.

[IS89] M. Iri and K. Sugihara. Construction of the Voronoi diagram for one million generators in single-precision arithmetic. In *Proceedings of the 1st Canadian Conference on Computational Geometry*, Montréal, 1989.

[Jac95] D. Jackson. Boundary representation modelling with local tolerances. In *Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 247–253, 1995.

[Jus92] N.P. Juster. Modelling and representation of dimensions and tolerances: a survey. *CAD*, 24(1):3–17, jan 1992.

[KLN91] M. Karasick, D. Lieber, and L.R. Nackmann. Efficient delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10:71–91, Jan. 1991.

[Knu92] D.E. Knuth. *Axioms and hulls.* Lecture Notes in Computer Science (606), Springer-Verlag, 1992.

[Mic87] D. Michelucci. *Les représentations par les frontières : quelques constructions; difficultés rencontrées (in french)*. PhD thesis, École Nationale Supérieure des Mines de Saint-Étienne, 1987.

[Mic95] D. Michelucci. An epsilon-arithmetic for removing degeneracies. In *Proceedings of the IEEE 12th Symposium on Computer Arithmetic*, Windsor, Ontario, July 1995.

[Mil88] V.J. Milenkovic. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic*. PhD thesis, Carnegie-Mellon, 1988.

[MM96] D. Michelucci and J-M. Moreau. Lazy arithmetic. *to be published in IEEE Transactions on Computers*, 1996.

[NSTY93] J. Nakagawa, H. Sato, K. Toshimitsu, and F. Yamagushi. An adaptive error-free computation based on the 4x4 determinant. *The Visual Computer*, 9:173–181, 1993.

[OTC87] G. Ottmann, G. Thiemt, and Ullrich C. Numerical stability of geometric algorithms. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, pages 119–125, 1987.

[PA94] R.M. Persiano and A. Apolinário. Boundary evaluation of csg models by adaptative triangulation. In *CSG 94 : Set Theoretic Solid Modelling Techniques and Applications*, Information Geometers Ltd, april 1994.

[SBE95] P. Stucki, J. Bresenham, and R. Earnshaw. Computer graphics in rapid prototyping technology. *IEEE Computer Graphics and Applications (special issue on Rapid Prototyping)*, 15(6):17–19, Nov. 1995.

[Seg90] M. Segal. Using tolerances to guarantee valid polyhedral modeling results. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):105–114, August 1990.

[Tau93] G. Taubin. An accurate algorithm for rasterizing algebraic curves. In *Second Symposium on Solid Modeling and Applications, ACM/IEEE*, pages 221–230, May 1993.

[Yam87] F. Yamagushi. Theoretical foundations for the 4x4 determinant approach in computer graphics and geometrical modeling. *The Visual Computer*, 3:88–97, 1987.